QUERG CHESS

John F. White

Wokingham
England

ABSTRACT

Extrapolating from several microcomputer implementations at moderate speed with correspondingly constraint depths of search and complexity of evaluation functions, the author is led to conjecture that there is *no* difference between
- deep search and superficial evaluation, and
- shallow search and complicated evaluation,
though deeper search may provide stronger resistance to human opponents.

1. INTRODUCTION

Many ideas for improvements to chess programs have been published over the years, but there are few comparative data about the techniques. Owing to fluctuations in programming skills, the reader is left to wonder whether alleged improvements in computer-chess heuristics may rather reflect superior programming ability by their authors.

I wish to describe my own results with chess programs while controlling one of the variables, namely the quality of the programmer. I have constructed, over several years, a series of chess programs called by the generic name of *Querg*. The programs are all implemented in 6502 assembly code on a 64K Atari 130XE domestic microcomputer, running at 2.2 MHz (in the UK; 1.8 MHz in the USA).

The latest variant, *SuperQuerg*, is claimed to be of comparable standard to the strongest of other programs currently available for the Atari 130XE computer. The name *Querg* has no meaning.

2. DESIGN CONCEPTS

It has been well established (Condon and Thompson, 1983) that the depth of computer search is important for the determination of program strength. Thompson found that a program with a deep search was consistently able to beat the same program playing with a shallower search. It is also widely acknowledged that a deep-search program is more effective at overcoming human opponents than a shallow-search alternative. What is less clear is whether a program would be better advised to allocate its search time to a complicated evaluation function with a shallow search, or to a deep search with a very simple evaluation, when playing another program or human opponent.

Positional Play

I am a moderately strong chess-player (ELO ca. 1800) with an emphasis on positional play, and I find that I can consistently beat even modern, deep-search commercial chess computers by a policy of playing a very slow game, taking no risks and waiting until the machine opponent makes a positional error, before steering for the endgame and winning. With this in mind, I designed *Querg* to play a slow positional game, to challenge my own chess play and to test whether the program would perform well against the deep-search commercial alternatives. The SCHACH program is also reported (Levy, 1983) to adopt a similar policy of complex evaluation and shallow search.

A serious problem with a complex evaluation function is that alpha-beta pruning becomes less efficient since no one move convincingly refutes another. It was apparent that some forcing lines would require purely tactical play. Therefore *Querg* is designed to handle turbulent positions at greater speed than the non-tactical ones.

3. PROGRAMMING QUERG

It is generally recognised that a chess program has three central features: *a move generator*, *a tree-search routine* and *an evaluation function* (EF). These are considered in turn.

3.1. Querg's Move Generator

Most versions of *Querg*, including the latest, use the well-known offset move generator (Frey, 1983; Levy, 1984, pp. 3-5), on a cylindrical board with a two-byte rim. Moves are generated and stored in a move list, one move list for all the moves at each ply of search.

Alternative Move Generators Considered

A 64-bit move generator (Cracraft, 1984) is not practical with an 8-bit computer, despite its undoubted advantage for computers with 64-bit processors. Table-driven move generation is actually slower (in my hands) than the offset generator if the tables are stored compactly, requiring the use of a complex system of pointers to the local head of each table for a given piece on a given square. If the tables are spaced at regular intervals, table-driven move generation is of similar speed to the offset generator, but requires excessive use of space in a small micro.

Future Move Generators

One method of move generation which has not been implemented in *Querg*, but will be in future variants, is based on the principle that the X and Y coordinates of any move on the board are stored in the upper and lower nibbles of one byte. Thus, square d5 is located as $45. The technique is coupled with an offset move generator without a rim for the board and is used in Wiereyn's mate-finding program (Wiereyn, 1985). This method confers certain advantages for the rapid detection of checks, see Section 3.

Incremental Move List

Incremental updating of move lists has not been very successful in my hands. Whether generated by the offset method or by tables, the moves for each piece are stored in a special table - one list of moves for each piece - and whenever a piece is moved, the move lists are updated only for the pieces affected by the move.

This method has the advantage that only a fixed amount of memory is needed to hold all the moves at any ply depth, whereas the earlier method required complete and separate move lists to be stored for each ply of search.

Sliding Pieces

The problem occurs with the sliding pieces (B, R, Q). When White opens a game with the move e2-e4, not only the Pawn, but also the Bishop on f1, and the Queen on d1, must all have their move lists updated. The affected sliding pieces can be found by either of two methods:

(a) A search through all the lists for both sides, locating all sliding pieces and then looking to see whether the specific list for each sliding piece needs to be updated. In order to accomplish this, pseudo-moves, including all moves which would be legal if a piece could move onto a square occupied by a piece of its own colour, need to be stored in the piece movelists.

(b) A search outwards from squares e2 and e4 to see whether any sliding piece (distinguished on the board by a unique token value) is encountered in the direction appropriate to the piece (Bishops / Queens on diagonals; Rooks / Queens on vertical / horizontal lines).

Discussions

Method (a) is slow and cumbersome, since it is necessary to generate, and then to reject, many pseudo-legal moves which would not be required by another method of move generation. Method (b) is a great deal faster, but is complicated to program. Moreover, a program which carries out a search to a given position at 4-ply will generate three separate move lists to reach the position by thc older procedure (generation of the moves for all pieces at each ply level), but will have carried out the equivalent of the generation of four move lists if the piece lists are incrementally updated, and the lists have been updated after each piece has been moved at the fourth ply.

Naturally, since most of the updating of piece lists will be carried out at the greatest depth of search, incremental updating of lists is slower under these conditions. If the program 'knows' that it will not need to search beyond 4-ply before updating its piece movelists, I find that method (b) becomes faster than the older method of move generation. Howevcr, the effect of not updating the piece movelists at the final ply depth is to deprive the EF of up-to-date information about square control and/or mobility, see Section 3.4.

3.2. Tree Search

Querg Chess uses the well-known method of iterative deepening to search the game tree. I have never tried a depth-first search. The moves are sorted into decreasing score order after the first iteration; thereafter, the order of moves is only changed by moving the 'best-yet-found' to the head of the list.

Killers

Two killer moves are 'remembered' and used to reject moves. Attempts to generate and verify a killer move before constructing a complete move list failed to provide any benefit in my hands. One of the advantages of a linked move list of all moves generated at each ply, compared with storing piece movelists, is that the moves may readily be used in any order. For this reason, all capturcs are tried as killer moves after the two original stored killers (Bettadapur, 1986).

Forcing Moves

Forcing moves, namely checks, replies to check, pawn promotions, threats of pawn promotion by the side not on the move, and captures, are all searched to a maximum depth of 3-ply above the current iteration, after which a swap-off algorithm (Levy, 1984, pp. 54-57) is implemented.

HIPL

A special routine *HIPL* (high-ply-pruning) scouts each move at any ply depth above the current iteration, and rejects moves which are not forcing as defined above. *HIPL* also provides a 'null-move' score in case the side on the move does not clect (or lacks the ability) to make a check or capture.

HIPL is very effective in saving time (some 25-30% on average) searching deeply into the tree, since it avoids the unnecessary sequence MOVE-EVALUATE-UNMOVE for non-forcing moves. The MOVE and UNMOVE routines are lengthy, updating the material score, handling castling and pawn promotion as well as providing the obvious functions. I have not seen a routine like *HIPL* described before, but would be surprised if it were novel.

The Search Strategy

Current versions of *Querg* do not use pruning, excepting that caused by the alpha-beta algorithm and *HIPL*. I spent some time exploring a tapcred search (i.e., fewer moves examined at each deeper iteration) and also with a variant of the 'extcnded razor' (Birmingham and Kent, 1977) from the root moves, but found little benefit. The razor pruning was based on my observation of my own style of chess play; if I found a strong move, I did not waste time looking for a better one. Moves were razored out from *Querg* on the basis that their root scores were much lower than a very good backed-up score for an earlier move, and the size of the cut-off was modified on the basis of how much time remained to the program. However, most of the time of search is spent on the first three to four moves at each iteration, so that pruning out subsequent moves is scarcely worth the effort.

I have found the effects of the alpha-beta window to be very variable. Current versions of *Querg* employ

(A) a window of 1.5 Pawns either way, with a feedover at each iteration. The window has been modified to adjust itself after each root move has been evaluated, an idea taken from the PVS routine described by Marsland (1986). The principal variation found at the end of each iteration is also 'fed over' into the next iteration.

I have also tried

(B) the option of widening the window if the score of the first move at an iteration fails to remain within the original window, the idea being that, if the best move at the head of the move list fails to stay within the window, it is unlikely that a subsequent move will be any better.

There is little practical difference seen overall between methods (A) and (B), although each appears to be much better than the other in some board positions.

3.3. Detection of Checks

Checks are detected by a variant of the method given by Wiereyn (1985), modified to suit a cylindrical representation of the chess board, and accordingly rather slower than the original described - the cylindrical 12 $\times$ 10 board is not well-suited to implementation of this procedure.

An alternative technique (Levy, 1984, p. 4) which I have explored is to ignore checks altogether, until one of the Kings has been captured, then to step down one ply and reject the move now under consideration, which must have been illegal (King in check). This method saves time when there are few checking lines from a given position, but is inefficient if the Kings are vulnerable to checks - even sacrificial checks.

If moves are stored in piece movelists, then it is also possible to note whether any piece is attacking the opposing King and to store that fact in the piece's move list. Separate counters are additionally kept of the number of times that each King is being attacked. After any move, the knowledge of whether either King is in check is instantly available. I have found this process, previously undocumented, to work very well for those versions of *Querg* which use piece movelists.

Scanning the Nodes

Querg's basic tree searching system, including checks and an evaluation of material balance only, but excluding nodes eliminated by *HIPL*, provides some 150 terminal evaluations per second on a 64K Atari 130XE. If *HIPL* moves are included, and ignoring checks until a King has been captured, the figure rises to some 450 terminal nodes per second.

Versions of *Querg* which employ piece movelists generate 100 terminal nodes per second if the lists are updated at each ply, or some 250 nodes per second if the lists are not updated at the deepest ply; in both cases excluding nodes eliminated by HIPL.

Machissimo Claims

The reader will be aware of the 'machissimo' claims of certain manufacturers of commercial chess programs for their machine's speed, measured in nodes per second. This is a misguided claim for the following reasons:

(a) Grandmasters boast about how few positions they examine, and depth of computer search merely reveals the inadequacy of the EF.

(b) The number of nodes examined may be dependent on several factors - see the figures quoted above for *Querg*, which depend on whether nodes discarded by *HIPL* should be taken into account. Again, omission of the much-used killer heuristic will increase the number of nodes per second examined by a program, since the killer takes time to implement, but will retard the speed at which the best move is found.

3.4. Evaluation Functions

The current version of *Querg* uses, largely, a first-order (Larson, 1987) evaluation process in which the EF is recalculated after every move for a full positional assessment. This technique is undoubtedly slow, but it is not practical to update many positional factors during the tree search. However, the material count and the turbulence factor are so updated, and the Pawn structure is matched to see whether it is unchanged from the original structure (in which case the early value is taken for the Pawn contribution).

Querg makes every effort to avoid making a full evaluation, and terminal nodes which are turbulent or fall outside the alpha-beta window (adjusted for pieces left *en prise*) are only scored for material (and Pawns, in the case of turbulent positions). The role of *HIPL* in restricting evaluation has already been described. This mixture of cursory and complicated evaluation provides *Querg* with a good combination of tactical (fast) and positional (slow) play, depending on the board position.

All evaluation functions essentially need to measure four items:

i) Material count
ii) Pieces left *en prise* (modifying (i))
iii) Mobility
iv) Pawn structure

It is easy to see that a complicated EF can assess all these factors accurately, but slowly. Alternatives for the material count involve keeping a tally of pieces taken during the tree search (*Querg* does so). Pieces left *en prise* can be detected by examining each 'terminal' node by a further ply of search, to see if the other side can make a capture. I have found this method to be rather slow in use, but it does "improve" the count of nodes per second evaluated.

Mobility

Mobility can be updated in a number of ways, although I have found none to be better for *Querg* than its present mix of fast / slow evaluation. When piece movelists are updated during the tree search, it is possible to store with each piece list a count of the number of moves. Each update then subtracts the old piece mobility from the total score and stores the new mobility in the list while adding it to the total. In addition, the updated piece lists can form the basis of a faster full evaluation of each terminal node - square control can be updated at the same time.

Another alternative is to provide *score tables* for pieces standing on particular board squares. Thus, for example, a Knight on square a1 may score zero, while on square e5 it may score +10 points. This method of 'mobility' assessment is very fast, although many tables have to be stored, including different tables for opening-, mid- and end-game play and for Pawns of both colours. It has also the merit that the tables provide a measure of the long-term mobility of a position. Thus, in mid-game, a Queen move from d1 to d4 may score favourably even though a conventional EF might bar such a move because of (say) temporarily blocked Pawns in the centre.

Pawn Positions

It is not practical to calculate, store and recall Pawn positions on a 64K microcomputer, despite the obvious appeal of this method. *Querg* uses a fast Pawn EF based simply on the number of Pawns in a column, with a slower, first order, more detailed calculation (when required) of Pawn structure around the King, backward Pawns and pushing passed Pawns.

Given the design constraints of a 64K Atari none of the strategies listed in this Section 3.4 essentially improve *Querg*'s fast and slow routines. For this reason, they have been abandoned.

3.5. Draws

Querg acknowledges draws by stalemate, by three-fold repetition of positions, by the 50-move rule and by insufficient mating material. The repetition of *positions* is recognized by the storage of up to 50 moves (thereby implementing the 50-move rule) coupled with a hash-coding system to examine positions at the root nodes only. The benefit of recognising draws by repetition of positions is quite considerable when playing a program which lacks this ability, and *Querg* has saved many games which were otherwise lost against such opposition as the *Morphy* program.

Early versions of *Querg* simply avoided draws caused by three-fold repetition of *moves* by both sides, but this proves to be inadequate when, as often occurs, one side or the other starts to triangulate moves:
1. Qg5-h6+ Kh8-g8 2. Qh6-g6+ Kg8-h8 3. Qg6-f6+ Kh8-g8 4. Qf6-g6+ Kg8-h8 5. Qg6-h6+ Kh8-g8 ... .

The current version of *Querg* has the additional characteristic that with sufficient mating material or with a major advantage it will sooner or later find a winning continuation even with a relatively simple EF.

3.6. Other Features

Time control on *Querg* is as described by Hyatt (1984). The program is interrupted as soon as the current time allotment is up, and no attempt is made to finish the search unless the program foresees a heavy loss of material, when it will continue to look for a better move. Time control has been a constant problem with *Querg*, until adoption of the present method, and I have not discovered any satisfactory scheme for halting the search at a time 'convenient' for the program.

The latest variant of *Querg* 'thinks' while it is awaiting its opponent's response. This facility, still relatively rare, is implemented by using the vertical-blank interrupt, provided by the Atari 130XE, to handle keyboard input, while the program continues to respond to its own suggested hint for its opponent's move. Interestingly, *Querg* is good at predicting its opponent's moves when attacking, but poor when defending. Doubtless this is a feature of all programs with a similar facility.

Querg terminates its *thinking-while-waiting* after double the normally allocated time. It will therefore sometimes make instant responses, allowing more time for other moves. In other cases, where *Querg* has successfully predicted its opponent's move, it can continue its search from the position which it has already reached.

Querg employs a *chopper* heuristic: where only one legal move exists, it will be made at once without search. There is an argument for restricting chopped moves to a search depth of 2-ply, since this will provide a hint for the program to consider while awaiting its opponent's move.

A method for providing *compact book opening* strings was described by K. Spracklen (1983). However, it is very difficult to alter the book once it has been set up. I use a series of book strings, grouped according to type and with a 'book window' employed by the program to narrow the number of lines examined as the depth of book search increases after each move in the opening. The book strings can then readily be modified with a word processor and poked into memory with a separate program written in Basic. The book window ensures that moves made from the book are found very quickly.

Querg is also fitted with a *mate-finder mode*, seeking mates in one to five moves. This is accomplished without any evaluation function at all, setting the alpha-beta window to within 1.5 of mate and raising each iteration by two ply. Only checks are now counted as forcing moves. This procedure is quite effective for finding mates quickly, but hardly compares for speed with the dedicated mate-finding programs such as that of Wiereyn (1985). Its speed on Grottling's (1985) problems is similar to that of the *Capablanca* chess module. However, I have observed, like Lindner (1985), that the speed of finding the solution is critically dependent on the order in which the moves are examined.

4. PERFORMANCE

I should like to be able to say that *SuperQuerg Chess* beats all opposition, human or machine. Unfortunately, this is not the case. However, tests between *Querg* and other programs running at 2 MHz (particularly those on the Atari 130XE micro) show that *Querg* is of a similar standard to the commercially-available opposition.

Querg tends either to beat *Parker "Chess"* (actually the 1983 *Cyrus* program) quickly by tactics, or to lose slowly to *Cyrus*' superior strategy (*Cyrus* is probably the finest in this respect of those commercially available). Conversely, *Querg* tends to lose quickly to Bryant's fast-searching *Colossus 3.0*, but to win on positional merit when it can survive the tactics.

I played a small tournament early in 1987 at 15 to 20 seconds per move (all vs. all, once as Black and once as White). The tournament was restricted to programs available with a 6502 microprocessor running at 2 MHz, and included *NovaQuerg* (1987), *Colossus 3.0* (1984), *Conchess* (1982), *Cyrus* (1983), *Morphy* (1981), *Sargon 2.5* (1980), *Steinitz* (1983) and *Super Enterprise* (1985). The time was chosen to point up the differences in playing styles, given the well-known law of diminishing returns from deep computer search as the time allowed increases.

Apart from *Sargon 2.5* with 2.5 points, only four points separated the top and bottom programs (from a maximum of 14 points) and the difference was not statistically significant. *NovaQuerg* scored 6.5 points. No program demonstrated a clear superiority, but the two nominal joint winners (on 9.5 points) comprised one of the strongest deep-search programs (*Super Enterprise*) and one of the shallowest searching, but highly positional, alternatives (*Cyrus*).

Querg's play reflects that of its creator, so that it provides stout opposition to any human attempt to grind it down positionally. However, it can be decisively crushed by a tactical player prepared to make sacrifices.

The position of Diagram 1 provides a good tactical test for any chess computer. Arising from an actual game, probably the only saving move for Black is f7-f5. The complexity of the position, and the veiled threat of White's Qd2-h6, combine to make finding the right move a difficult task.

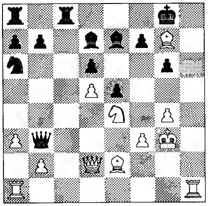

DIAGRAM 1
Black to move.

Program	Approx. time (mins : secs) to find f7-f5	Iteration
SuperQuerg (1988)*	4 : 15	3
NovaQuerg (1987)*	7 : 00	3
Colossus 3.0 *	2 : 45	4
Conchess (1982)	12 : 00	4
Cyrus (1983)*	12 : 00	4?
Morphy	22 : 00	4?
Steinitz	1 : 30	3
Super Enterprise	1 : 30	3

*runs on Atari 130XE microcomputer. Otherwise stand-alone machine.

5. CONCLUSION

It is impossible to make exact comparisons between programs using different playing styles on different hardware. Nevertheless, on the basis of extensive trials between variants of *Querg* and a range of other commercially available chess programs of different styles running at 2 MHz, I conjecture that there is little practical difference to perceived playing strength between programs (running on similar hardware) whether the allotted time is spent on a deep search and a superficial evaluation, or on a complicated evaluation and a shallow search.

However, I concur with the view that the deep search alternative provides the stronger opponent to human opposition. This difference may reflect the fact that it takes less in the way of artificial intelligence (the evaluation function) to fool a deep-searching chess program than is required to overcome the superb pattern-searching ability of the human mind.

6. REFERENCES

Bettadapur, P. (1986). Influence of Ordering on Capture Search. *ICCA Journal*, Vol. 9, No. 4, pp. 180-188.

Birmingham, J.A. and Kent, P. (1977). Tree-Searching and Tree-Pruning Techniques. *Advances in Computer Chess 1* (Ed. M.R.B. Clarke), pp. 89-97. Edinburgh University Press, Edinburgh .

Condon, J.H. and Thompson, K. (1983). Belle. *Chess Skill in Man and Machine* (Ed. P.W. Frey), 2nd edition, pp. 201-210. Springer-Verlag, New York.

Cracraft, S.M. (1984). Bitmap Move Generation in Chess. *ICCA Journal*, Vol. 7, No. 3, pp. 146-153.

Frey, P.W. (1983). An Introduction to Computer Chess. *Chess Skill in Man and Machine* (Ed. P.W. Frey), 2nd edition, pp. 54-81. Springer-Verlag, New York.

Grottling, G. (1985). Problem-Solving Ability Tested. *ICCA Journal*, Vol. 8, No. 2, pp. 107-110.

Hyatt, R.M. (1984). Using Time Wisely. *ICCA Journal*, Vol. 7, No. 1, pp. 4-9.

Larsson, J.E. (1987). Challenging that Mobility is Fundamental. *ICCA Journal*, Vol. 10, No. 3, pp. 139-142.

Levy, D.N.L. (1983). *Computer Gamesmanship,* pp. 128-131. Century London.

Levy, D.N.L. (1984). *The Chess Computer Handbook.* Batsford, London.

Lindner, L. (1985). A Critique of Problem-Solving Ability. *ICCA Journal*, Vol. 8, No. 3, pp. 182-185.

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19.

Spracklen, K. (1983). Tutorial: Representation of an Opening Book Tree. *ICCA Newsletter*, Vol. 6, No. 1, pp. 6-8.

Wiereyn, P.H. (1985). Inventive Problem Solving. *ICCA Journal*, Vol. 8, No. 4, pp. 230-234.

NOTES

SOME COMMENTS CONCERNING AN ARTICLE BY DE GROOT

I. J. Good

Virginia Polytechnic Institute
and State University
Virginia, USA

I was interested in A.D. de Groot's article "Some Benefits of Advances in Computer Chess" in the *ICCA Journal* for June 1987. He made a strong case for computer chess but the benefits for psychology, philosophy, and even for pseudognostics (A.I.) would be even greater if some competitions were held in which no more than 1000 positions could be examined at any one move. This would encourage the writing of programs depending more on intellect and less on memory. Unfortunately the limitation would be difficult to enforce in competitions but a step in the right direction would be to organize competitions for computers playing "randomized chess". In randomized chess the initial positions of the pieces on ranks 1 and 8 are randomized while preserving white-black symmetry, and there are a few other minor rules. This point was emphasized in Appendix J of my article "A five-year plan for automatic chess" (*Machine Intelligence II*, proceedings of a workshop in Edinburgh in June, 1966, E. Dale & D. Michie, eds., Oliver & Boyd, 1968, 89-118). Am I wrong in thinking that the suggestion is still being ignored after twenty years? I am not suggesting that the present form of competition should be abandoned, only that it should be supplemented.

For randomized chess De Groot's pioneering work on the way chess-players think would become even more relevant than it is at present.

To come to another point, in De Groot's article (page 75) he quotes an estimate of 10^{11} by Jongman (1968?) for an upper limit to the number of positions possible after Black's 20th move in a game played at master strength throughout. The lower and upper estimates in my article, page 109, were $10^{8.5}$ and $10^{11.5}$. Strictly, these were estimates of the number of *lines* up to that point; the number of *positions* reached is a little less because of transpositions. It is fair to say that my estimate of the upper limit is essentially the same as Jongman's. I also gave estimates for durations other than twenty moves, for example, between fifty thousand and one million lines after ten moves, and between three trillion and one hundred quadrillion after thirty moves.